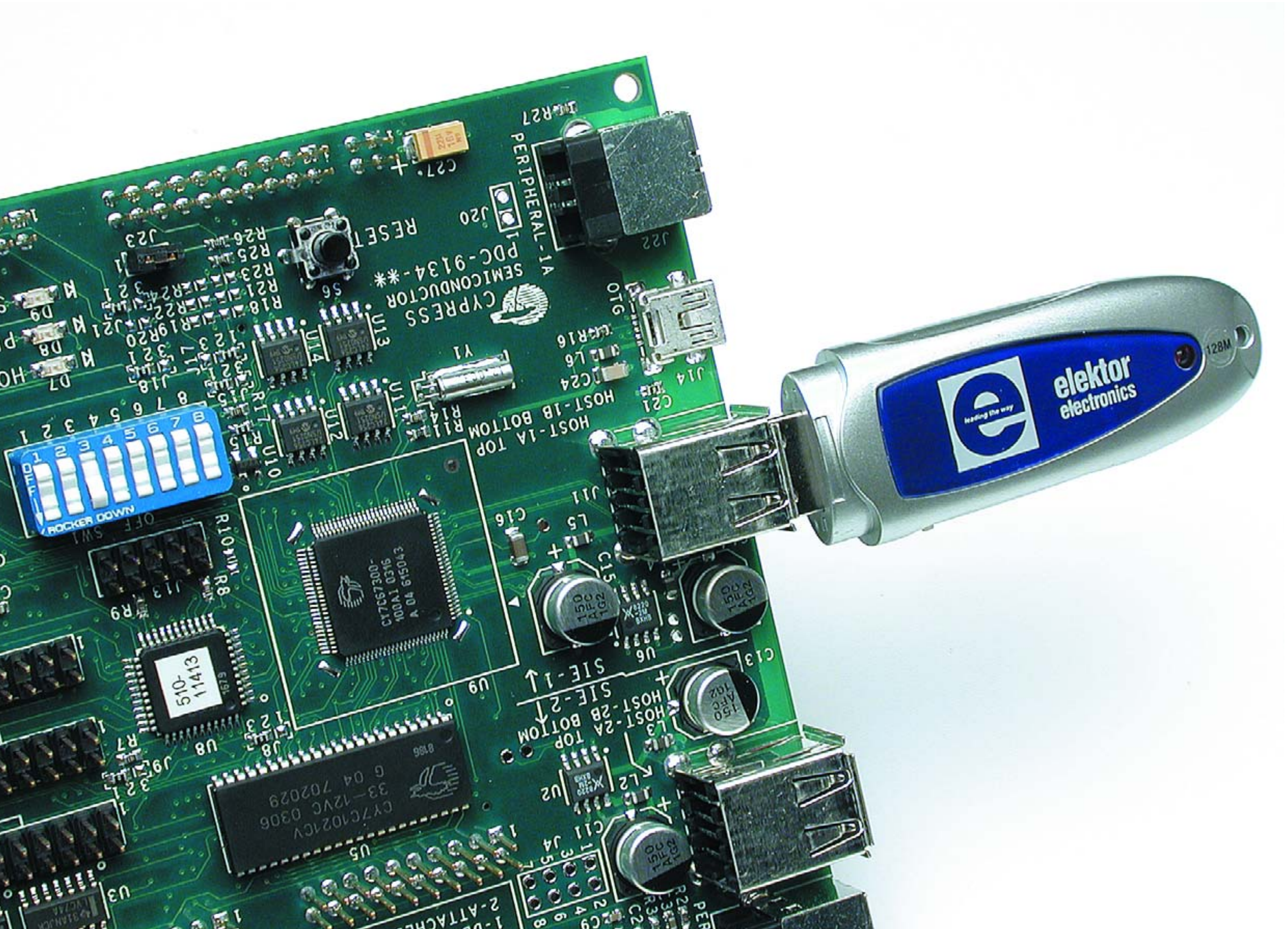
USB EMBEDDED H

for removable mass storage devices

Mark Schultz
Senior Applications Engineer - Cypress Semiconductor

Till now USB has been available only on personal computers, leaving large, untapped potential in the area of embedded systems. New embedded host controllers from Cypress herald a new era.



OST CONTROLLER

An embedded system is defined as hardware and firmware — either stand-alone or part of a larger system — usually with some sort of operating system. The operating system can be Windows CE, VxWorks or a more simple system consisting of 'home made' code. Using the above definition, it can be said that any electronic device that has a processor has the potential to be an embedded USB host.

Designs that incorporate a USB Host controller have a distinct advantage over traditional designs in the sense that they can now host any type of USB device. The obvious advantage here is that storage space for a design can be dynamically added or removed at any time. Consider the case of an MP-3 player that contains a USB Host. MP-3 files can be easily downloaded via a USB flash drive or any other media that includes USB support. Many other types of applications should be able to realize similar performance with the addition of a Host USB solution, such as the ability to perform field upgrades or download critical data to a non-networked system.

Existing solutions for expansion of embedded systems usually involve a laptop or some other type of portable computing device connected to the embedded system via a serial cable. Although this is a simple and effective solution, it requires the extra PC and additional cabling. Additionally, there could be problems associated with the use of legacy ports.

Typically, a PC is a USB host. Anything else containing USB is a peripheral. As an example, take the case of a digital camera. A digital camera has traditionally been a USB peripheral. To print an image to a USB based printer, a PC is first used to upload the images from the camera to the PC and then to download them to a printer. If the digital camera had USB host capabilities, it could download the images directly to the printer.

Of course, such functionality comes with a price. A USB Host is responsible for a number of tasks including enumeration of devices, task scheduling and bandwidth allocation. Fortunately, there will be a code framework that will handle these tasks for us.

This article will outline some of the basics required for interfacing a USB Host controller to a removable mass storage device. A short primer on the use of SCSI commands and how they relate to file systems will be presented as well as a quick overview of the FAT file system. This will be followed by a design example.

Mass storage basics

Communication via the USB interface uses SCSI commands embedded inside a UFI Packet. A file system is a logical structure used to track various parameters of the storage media.

FAT File System

This design will support the File Allocation Table (FAT) type of file system. In the FAT file system, there is a table that contains a number corresponding to each section or cluster on the disk. It will be necessary to use the File Allocation table to locate the individual clusters within a file since they may not all be contiguous. There are other types of file systems such as NTFS (NT File System) and UFS (Unix File System) that will not be supported by this design.

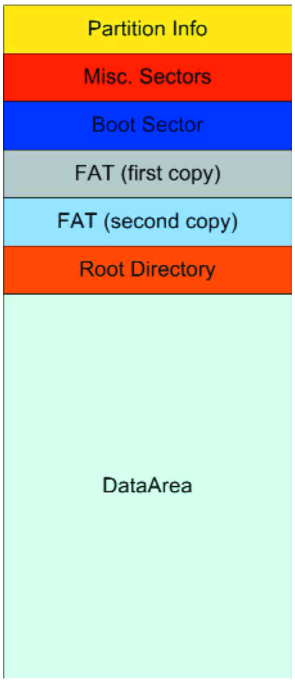


Figure 1. File structure system.

Our file system will be logically represented by a data structure called a volume. A volume contains information about the location of the File Allocation Table (FAT), root directory, and a data area that is subdivided into sectors and clusters.

All mass storage devices contain a boot record. Aside from containing code that can be used for bootstrapping the device, this area also contains other useful information about the device that can be used to fill in the volume information. There is also a partition record for each useable partition on the device. We will use this later on as well.

The basic outline of the file system is shown in **Figure 1**. There are a number of distinct areas on a mass storage device beside just the directory and data area. The partition information begins at offset $1BE_H$ of the first physical sector on the drive. After this sector, there are a number of reserved sectors, followed by the actual file allocation table. The FAT operates on groups of sectors called clusters. The FAT contains information about the physical clusters on the drive such as whether they are available to be written to and where the next cluster in the file is located. There are actually two copies of the FAT per drive for redundancy purposes. The root directory is 512 entries long in a FAT16 system with each entry occupying 32 bytes. Thus the total root directory takes up 32 sectors. After that is the data area containing files and subdirectories. The start of the data area is said to be at cluster number 2, although in reality it is actually more than 512 sectors from the beginning of the drive. **Table 1** shows the volume information for one particular USB flash device: The beginning of the actual data is 235_H sectors from the beginning of the drive. The area before this con-

Table 1. Volume information

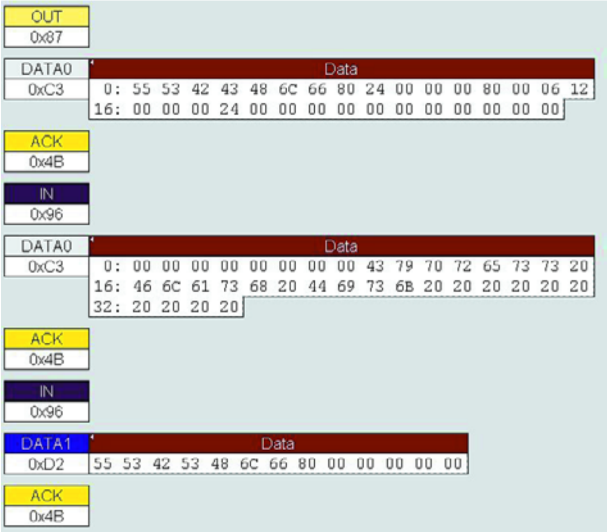
Volume Information	Data Size	Value for current drive
Fat Size	BYTE	FA_H
Total Sectors	DWORD	$3E7E0_H$
Total Data Sectors	DWORD	$3E5c8_H$
Number of Clusters	WORD	$F972_H$
Root Dir Sectors	BYTE	20_H
First Data Sector	WORD	235_H
Bytes Per Sector	WORD	200_H
Sectors Per Cluster	BYTE	004_H
First FAT Sector	BYTE	21_H
First Dir Sector	WORD	215_H

tains the partition record, boot record, two copies of the FAT, and the root directory. This information will be used later on to locate files on the drive.

SCSI

Many mass storage devices use SCSI commands for communication between the host and the device. We will be using three SCSI commands to perform our file operations. The commands are Inquiry, Read Capacity and Read(10). The format of the three SCSI commands to be used in this design is shown in **Table 2**.

Each command takes an Op Code and a Logical Unit Number (LUN). Some commands have an LBA field. An LBA is a Logical Block Address. A Logical Block is the smallest addressable section of a mass storage device, often referred to as a sector. Later on, we will see how to use an LBA to access file data. The Read (10) command also has a field for transfer length. Transfer length generally refers to the number of sectors to be sent or received. For the purpose of simplicity, we will work with only one Logical Unit. Thus, the LUN field will always be '0'.



```
OUT
0x87

DATA0   Data
0xC3     0: 55 53 42 43 48 6C 66 80 24 00 00 00 80 00 06 12
        16: 00 00 00 24 00 00 00 00 00 00 00 00 00 00 00

ACK
0x4B

IN
0x96

DATA0   Data
0xC3     0: 00 00 00 00 00 00 00 00 43 79 70 72 65 73 73 20
        16: 46 6C 61 73 68 20 44 69 73 6B 20 20 20 20 20 20
        32: 20 20 20 20

ACK
0x4B

IN
0x96

DATA1   Data
0xD2    55 53 42 53 48 6C 66 80 00 00 00 00 00

ACK
0x4B
```

Figure 2. UFI transaction using CBW and CSW.

UFI

Removable media mass storage devices containing a USB interface use a common interface known as the UFI interface. The SCSI commands are embedded in the UFI packets. A UFI transfer is carried out in three phases: *command*, *data* and *status*. The command phase employs the *Command Block Wrapper* (CBW) and the status phase, a *Command Status Wrapper* (CSW).

The CBW is a 31-byte packet with the first 15 bytes containing four bytes of signature and four bytes of tag, three bytes of commands and flags and 4 bytes for the length of the actual data to be read or written. The final 16 bytes comprise the command block, which is the actual SCSI command. The CBW signature is used to identify the device and is hard-coded to 55 53 42 43 by the UFI specification. Interestingly, the signature in ASCII is 'USBC' for a USB Command. The tag can be any 4 byte value. The CSW will use the tag value received during the CBW phase to report the status of the CBW.

The CSW is a 13-byte packet and is similar to the CBW in that it contains four bytes of signature hard-coded to 55 53 42 53 which, in ASCII is 'USBS' for USB Status. There is also a 4-byte tag that will be matched against the tag of the CBW. The CSW also contains four bytes for the residue, which is the expected transfer length minus the actual transferred data length. There is one additional byte used to report transfer status. An example of a USB transfer using a CBW and a CSW is shown in **Figure 2**.

The first transaction is the CBW. The signature is contained in bytes 0 through 3 and the tag is contained in bytes 4 through 7. The tag can be any random 32-bit number. The CSW will use the tag value to Acknowledge completion of the command specified in the CBW. The '24' in byte 8 is a hexadecimal value representing 36 decimal and is the actual length of the transferred data in the second phase.

The second transaction is the data stage and includes the 36 bytes of data read back from the USB mass storage device. In this case, the data is in response to an Inquiry command (12_H) that can be seen in byte 15 of the first transaction. Converting the 36 bytes of data beginning at the ninth byte to ASCII will reveal the string 'Cypress Flash Disk'.

The third transaction is the CSW. Note the match between the tags of the CBW and the CSW. There is no remaining data to be transferred as shown in bytes 8 through 11. Byte 12 contains the status of the transaction. For this field, a '0' denotes success.

The Cypress EZ-Host controller

The Cypress EZ-Host is a 16-bit RISC based microcontroller, which contains two Serial Interface Engines (SIEs).

Each SIE controls two USB host ports or one USB peripheral port. The EZ-Host also contains a number of peripherals such as RS-232, SPI, HPI and HSS. There are 16 K bytes of internal memory space and provisions for an external memory interface.

The EZ-Host includes a firmware framework that takes care of most of the USB Host details. With full support for various types of USB transfers, reading and writing to USB mass storage devices is as simple as filling in some information and letting the frameworks handle the rest.

The EZ-Host framework contains all of the necessary firmware to implement USB Host functionality. Included in this are task scheduling, device enumeration, bandwidth allocation and power management.

Applications, on the other hand, are the firmware to control specific USB devices and communicate their data to an end application.

At the heart of the frameworks is the TD processor. The TD processor operates on a data structure called a Task Descriptor (TD) using its information to communicate with the USB hardware, specifically the Serial Interface Engine (SIE). It is important to note that each of the SIEs controls two ports and that there is a TD processor for each SIE.

The framework code incorporates a simple round-robin scheduler. The scheduler handles a number of tasks such as checking for device insertion and removal, checking the UART for incoming messages and executing the 'run' function of every installed device driver.

Basic USB transfers are used to move data in either direction (Host to Peripheral and Peripheral to Host). The host initiates the transfers. To move data to a peripheral, the host uses the function usb_send_bulk_out. To get data from a peripheral, the host uses usb_recv_bulk_in. Each of these functions requires some additional data – a pointer to the device structure, the USB endpoint to be used, the length of transfer, and a pointer to the send or receive buffer. Once these details are specified, the framework takes care of the rest.

Framework Flow

The framework code will execute as follows. On power on reset, the EZ-Host microprocessor will initialize all registers and counters as well as all device structures. It will then enter a loop:

- Check the host USB ports for any changes in status (devices inserted or removed).
- Check the TD processor and get status of all TDs running on the two SIEs.
- Go through the list of active device drivers and execute the run function for each.

Checking the host USB port for status changes requires inspection of a change variable. The port change interrupt handler will set this variable if a change occurs. If the port shows a change, enumeration code will be executed to service this change.

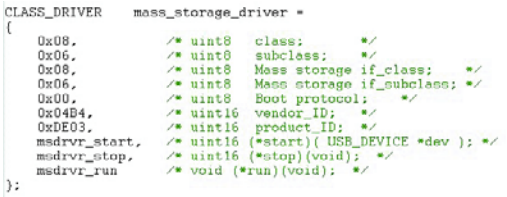
If a device is found, frameworks code will attempt to match the device to a registered driver. Driver matching can be accomplished in a number of ways. If there is only one specific device to be supported, the device's VID and PID can be added to a 'Targeted Peripheral List'. The device's VID and PID are then checked against this list. A more common method of device/driver matching is via device class and interface class. In the case of mass storage devices, both the device class and interface class have a value of '8' and their respective subclasses have a value of '6.' So, when the driver is created, these values will be entered into the appropriate fields (see **Figure 3**).

Table 2. SCSI commands

SCSI command	Inquiry	Read Capacity	Read(10)
Byte #0	12_H	25_H	28_H
Byte #1	LUN	LUN	LUN
Byte #2	Page Code	LBA (MSB)	LBA (MSB)
Byte #3	Reserved	LBA	LBA
Byte #4	Allocation Length	LBA	LBA
Byte #5	Reserved	LBA (LSB)	LBA (LSB)
Byte #6	Reserved	Reserved	Reserved
Byte #7	Reserved	Reserved	Transfer Length (MSB)
Byte #8	Reserved	Reserved	Transfer Length (LSB)
Byte #9	Reserved	Reserved	Reserved
Byte #10	Reserved	Reserved	Reserved
Byte #11	Reserved	Reserved	Reserved

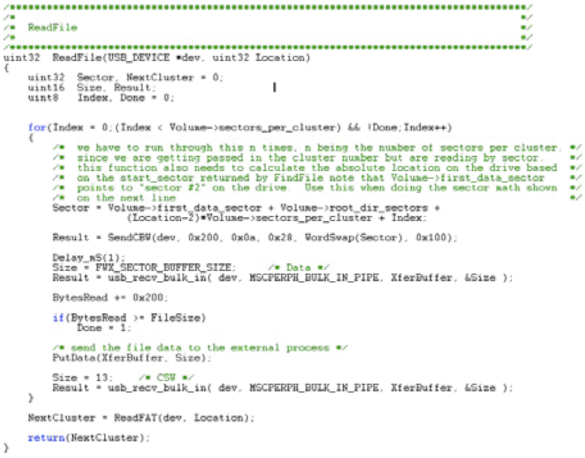
Building an application

In this section we'll show how to build a simple embedded USB design that controls a USB Flash drive. First, we will need to create a driver for the mass storage device. We can call the driver the mass_storage_driver or some name indicative of the capabilities of the device. The most important fields in this data structure are the class and subclass fields. The framework will use these values to match this driver up with any Flash device with the same class and subclass values. One other possibility here is to enter a specific Vendor ID and Product ID in the driver template. In this case, only the device containing



```
CLASS_DRIVER    mass_storage_driver =
{
    0x08,           /* uint8   class;        */
    0x06,           /* uint8   subclass;     */
    0x08,           /* uint8   Mass storage if_class;    */
    0x06,           /* uint8   Mass storage if_subclass; */
    0x00,           /* uint8   Boot protocol;    */
    0x04B4,         /* uint16  vendor_ID;    */
    0xDE03,         /* uint16  product_ID;   */
    msdrvr_start,   /* uint16 (*start)( USB_DEVICE *dev ); */
    msdrvr_stop,    /* uint16 (*stop)(void);  */
    msdrvr_run      /* void (*run)(void);  */
};
```

Figure 3: Device Driver Template.



```
/*************************************************************************/
/*                                                                       */
/*  ReadFile                                                             */
/*                                                                       */
/*************************************************************************/
uint32  ReadFile(USB_DEVICE *dev, uint32 Location)
{
    uint32  Sector, NextCluster = 0;
    uint16  Size, Result;
    uint8   Index, Done = 0;

    for(Index = 0;(Index < Volume->sectors_per_cluster) && !Done;Index++)
    {
        /*  we have to run through this n times, n being the number of sectors per cluster.  */
        /*  since we are getting passed in the cluster number but are reading by sector.     */
        /*  this function also needs to calculate the absolute location on the drive based  */
        /*  on the start_sector returned by FindFile note that Volume->first_data_sector     */
        /*  points to "sector #2" on the drive.  Use this when doing the sector math shown  */
        /*  on the next line                                                                 */
        Sector = Volume->first_data_sector + Volume->root_dir_sectors +
                     (Location-2)*Volume->sectors_per_cluster + Index;

        Result = SendCBW(dev, 0x200, 0x0a, 0x28, WordSwap(Sector), 0x100);

        Delay_mS(1);
        Size = FWX_SECTOR_BUFFER_SIZE;      /* Data */
        Result = usb_recv_bulk_in( dev, MSCPERPH_BULK_IN_PIPE, XferBuffer, &Size );

        BytesRead += 0x200;

        if(BytesRead >= FileSize)
            Done = 1;

        /* send the file data to the external process */
        PutData(XferBuffer, Size);

        Size = 13;    /* CSW */
        Result = usb_recv_bulk_in( dev, MSCPERPH_BULK_IN_PIPE, XferBuffer, &Size );
    }
    NextCluster = ReadFAT(dev, Location);

    return(NextCluster);
}
```

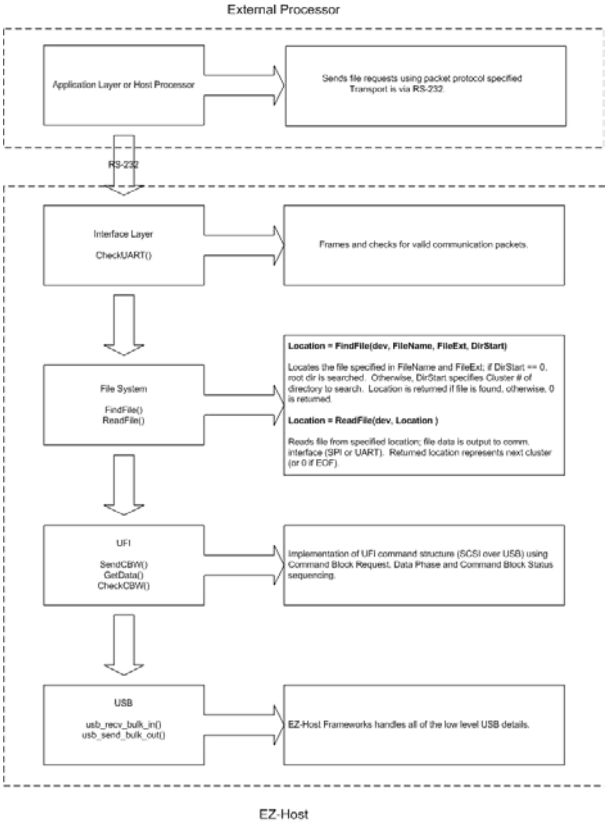
Figure 4. ReadFilr Code.

those Vendor and Product IDs will be started when plugged in.

Once the driver is created, the next steps are to add the start, stop and run functions plus the class codes for the device so the driver can be found. A driver structure is shown in **Figure 4**.

The name of the driver function has to be added to the file drvrlist.h:

```
#define FWX_DRIVER_LIST  { &mass_storage_driver }
```



Sends file requests using packet protocol specified
Transport is via RS-232.

Frames and checks for valid communication packets.

Location = FindFile(dev, FileName, FileExt, DirStart)

Locates the file specified in FileName and FileExt; if DirStart == 0, root dir is searched. Otherwise, DirStart specifies Cluster # of directory to search. Location is returned if file is found, otherwise, 0 is returned.

Location = ReadFile(dev, Location)

Reads file from specified location; file data is output to comm. interface (SPI or UART). Returned location represents next cluster (or 0 if EOF).

Implementation of UFI command structure (SCSI over USB) using Command Block Request, Data Phase and Command Block Status sequencing.

EZ-Host Frameworks handles all of the low level USB details.

Figure 5: Simple USB Host Stack.

The mass_storage_driver's 'start' function will use the CBW/CSW protocol to interrogate the device to determine its characteristics. Code inside of these functions will get the data and pass it on to the application layer of the design. This will enable the higher-level layers of the application to handle the specifics of file management. The stop function will clean up any pending operations or transfers and notify the application layer that a device has been removed. The run function, finally, will be used to check for any incoming commands from the application code.

Using the frameworks to read a USB Flash drive

Now it is time to create some application code that actually does something. At this point, we have created a driver and have a means to match it up to a device by class and subclass matching. Next up is to add code to the driver for its start and run functions.

When the driver starts, we want it to go out and interrogate the flash device to gather information about its file system. Then, the run function will be used to periodically check as to whether the user wants to read any file data. The driver's start function will gather data about the file system in the following way:

- Perform a SCSI Inquiry command to get information on the attached device.
- Perform a SCSI Read Capacity command to get information about the size of the attached device.
- Perform a SCSI Read(10) with the LBA set to 0 to get data from the first physical sector on the flash device.
- Type cast the 16 bytes beginning at offset $1BE_H$ of the first sector to a partition record entry.
- Using the partition record information, locate the 'Start LBA' — the number of the cluster containing the boot record.
- Perform a SCSI Read(10) with the LBA value set to the Start LBA to read the sector containing the boot record. Use the information in the boot record to fill in the volume information.

At this point, we have used three SCSI commands and our knowledge of the drive layout to obtain the information that was presented in Table 1. We can now use this information to read any file on the drive.

The directory entry for a file will contain its Start Cluster number. This value will be passed to the file reading function in the Location field. The ReadFile function will read each sector in the cluster, then check the FAT to see if there are additional clusters for the file. The code required to perform a file read is shown in **Figure 4**. The Sector value can be calculated using information contained in the Volume structure. Note that the file reading loop executes once for each sector in the cluster. When a cluster has been fully read, the FAT is checked to find the NextCluster value. Also note the use of CBWs and CSWs for communicating with the drive.

Application Programming Interface

Figure 5 shows the hierarchy of the EZ-Host firmware. An external processor or application can communicate with the interface layer via RS232. Application code can be written to provide an API between the interface layer and the FindFile and ReadFile functions.
One possible interface layer is shown in **Figure 6**. Here, the UART is checked for a one byte Op Code followed by a filename, file extension and directory start location. An Op Code of '1' causes the FindFile function to be executed. An Op Code of '2' signifies a file read operation.

The FileName and FileExt fields in the packet specify the file to be found or read. The additional packet field, DirStart allows for the location of files in either the root directory (DirStart = 0) or in a sub directory (DirStart = start cluster of the subdirectory).

As an example, suppose we wanted to read a file called *File1* in the root directory:
The first UART Packet sent to the interface layer would have the parameter FileName set to *File1*, a DirStart value of '0' and an Op Code of '2' for a file read. The EZ-Host firmware would return the contents of the file *File1*.

Now, suppose we wanted to read a file called *File2* in the subdirectory *SubDir1*.

The first UART Packet would have the **FileName** parameter set to ***SubDir1***, the Op Code set to '1' for file find and the DirStart value set to '0' The FindFile function will return the starting cluster of the *SubDir1* directory to the application. Next, another UART packet would be sent, this time with the FileName parameter set to *File2*, the Op Code set to '2' for file read and the DirStart value set to the starting cluster of *SubDir1* which was returned from the previous FindFile function. At this point, the EZ-Host will return the contents of *File2*.

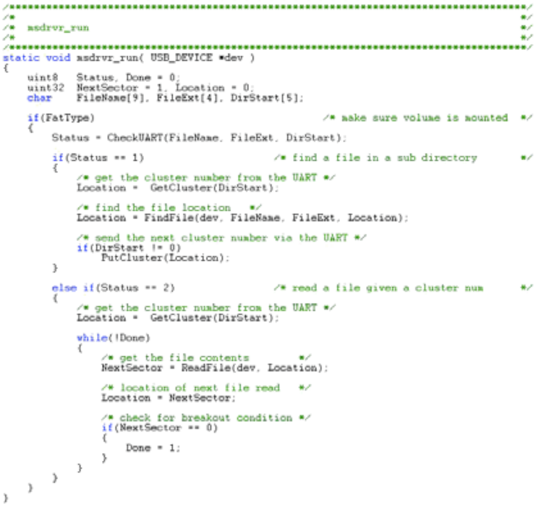
Using this API and the above examples, a file from any directory on the drive can be read.

Conclusion

Using an embedded Host USB controller, removable mass storage can be added to any design. This feature allows for downloading of critical data or for firmware upgrades to the design via USB.

An embedded USB host may be implemented with just one additional IC and no third-party software. An embedded USB Host framework has taken care of all of the lower-level details for us. Cypress' EZ-Host USB host can support the full speed Bulk transfers required to communicate with mass storage devices. The present code size is approximately 13 kbytes. File system support can easily be added to support application specific devices.

(040238-1)



```
/*************************************************************************************/
/*                                                                                   */
/*  msdrvr_run                                                                       */
/*                                                                                   */
/*************************************************************************************/
static void msdrvr_run( USB_DEVICE *dev )
{
    uint8   Status, Done = 0;
    uint32  NextSector = 1, Location = 0;
    char    FileName[9], FileExt[4], DirStart[5];

    if(FatType)                                    /* make sure volume is mounted  */
    {
        Status = CheckUART(FileName, FileExt, DirStart);

        if(Status == 1)                    /* find a file in a sub directory       */
        {
            /* get the cluster number from the UART */
            Location =  GetCluster(DirStart);

            /* find the file location   */
            Location = FindFile(dev, FileName, FileExt, Location);

            /* send the next cluster number via the UART */
            if(DirStart != 0)
                PutCluster(Location);
        }

        else if(Status == 2)               /* read a file given a cluster num      */
        {
            /* get the cluster number from the UART */
            Location =  GetCluster(DirStart);

            while(!Done)
            {
                /* get the file contents        */
                NextSector = ReadFile(dev, Location);

                /* location of next file read   */
                Location = NextSector;

                /* check for breakout condition */
                if(NextSector == 0)
                {
                    Done = 1;
                }
            }
        }
    }
}
```

Figure 6: Example Interface Layer.

Web pointers

FAT file system:
www.microsoft.com/hwdev/download/hardware/fatgen103.pdf

UFI Command Specification:
www.usb.org/developers/devclass_docs/usbmass-ufi10.pdf

USB Mass Storage Class Bulk Only Transport Specification:
www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf

Mark Schultz

Is a Senior Customer Design Center Engineer with Cypress Semiconductor. Mark holds a BSc in Information Technology from New Jersey College. Before arriving at Cypress, he worked on CCD camera bus interface boards at Princeton Instruments. Mark is a member of IEEE and has over 20 years experience in the electronics industry.